An Approach to Prioritize Classes in a Multi-objective Software Maintenance Framework

An Approach to Prioritize Classes in a Multi-objective Software Maintenance Framework

Michael Mohan and Des Greer
Department of Electronics, Electrical Engineering and Computer Science,
Queen's University Belfast, Northern Ireland, U.K.
mmohan03@qub.ac.uk, des.greer@qub.ac.uk

Keywords: Search based Software Engineering, Maintenance, Automated Refactoring, Refactoring Tools, Software Quality, Multi-objective Optimization, Genetic Algorithms.

Abstract: Genetic algorithms have become popular in automating software refactoring and an increasing level of attention is being given to the use of multi-objective approaches. This paper investigated the use of a multi-objective genetic algorithm to automate software refactoring using a purpose built tool, MultiRefactor. The tool used a metric function to measure quality in a software system and tested a second objective to measure the importance of the classes being refactored. This priority objective takes as input a set of classes to favor and, optionally, a set of classes to disfavor as well. The multi-objective setup refactors the input program to improve its quality using the quality objective, while also focusing on the classes specified by the user. An experiment was constructed to measure the multi-objective approach against the alternative mono-objective approach that does not use an objective to measure priority of classes. The two approaches were tested on six different open source Java programs. The multi-objective approach was found to give significantly better priority scores across all inputs in a similar time, while also generating improvements in the quality scores.

1 INTRODUCTION

Search-Based Software Engineering (SBSE) has been used to automate various aspects of the software development cycle. Used successfully, SBSE can help to improve decision making throughout the development process and assist in enhancing resources and reducing cost and time, making the process more streamlined and efficient. Search-Based Software Maintenance (SBSM) is usually directed at minimizing the effort of maintaining a software product. An increasing proportion of SBSM research is making use of multi-objective optimization techniques. Many multi-objective search algorithms are built using genetic algorithms (GAs), due to their ability to generate multiple possible solutions. Instead of focusing on only one property, the multi-objective algorithm is concerned with a number of different objectives. This is handled through a fitness calculation and sorting of the solutions after they have been modified or added to. The main approach used to organize solutions in a multi-objective approach is Pareto. Pareto dominance organizes the possible solutions into different non-domination levels and further discerns between them by finding the objective distances between them in Euclidean space.

In this paper, a multi-objective approach is created to improve software that combines a quality objective with one that incorporates the priority of the classes in the solution. There are a few situations in which this may be useful. Suppose a developer on a project is part of a team, where each member of the team is concerned with certain aspects of the functionality of the program. This will likely involve looking at a subset of specific classes in the program. The developer may only have involvement in the modification of their selected set of classes. They may not even be aware of the functionality of the other classes in the project. Likewise, even if the person is the sole developer of the project, there may be certain classes which are more risky or more recent or in some other way more worthy of attention. Additionally, there may be certain parts of the code considered less well-structured and therefore most in need of refactoring. Given this prioritization of some classes for refactoring, tool support is better em-

Mohan, M. and Greer, D.
An Approach to Prioritize Classes in a Multi-objective Software Maintenance Framework.
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 215-222
ISBN: 978-989-758-300-1

ployed with refactoring directed towards those classes.

Another situation is that there may be some classes considered less suitable for refactoring. Suppose a developer has only worked on a subset of the classes and is unsure about other areas of the code, they may prefer not to modify that section of the code. Similarly, older established code might be considered already very stable, possibly having been refactored extensively in the past, where refactoring might be considered an unnecessary risk. Changing code also necessitates redoing integration and tests and this could be another reason for leaving parts of the code as they were. There may also be cases where "poor quality" has been accepted as a necessary evil. For example, a project may have a class for logging that is referenced by many other classes. Generally, highly coupled classes are seen as negative coding practices, but for the purposes of the project it may be deemed unavoidable. In cases like this where the more unorthodox structure of the class is desired by the developer, these classes can be specified in order to avoid refactoring them to appease the software metrics used. However, we do not want to exclude less favoured classes from the refactoring process since an overall higher quality codebase may be achieved if some of those are included in the refactorings.

We propose that it would be helpful to classify classes into a list of "priority" classes and "non-priority" classes in order to focus on the refactoring solutions that have refactored the priority classes and given less attention to the non-priority classes. The priority objective proposed takes count of the classes used in the refactorings of a solution and uses that measurement to derive how successful the solution is at focusing on priority classes and evading non-priority classes. The refactorings themselves are not restricted so during the refactoring process the search is free to apply any refactoring available, regardless of the class being refactored. The priority objective measures the solutions after the refactorings have been applied to aid in choosing between the options available. This will then allow the objective to discern between the available refactoring solutions. In order to test the effectiveness of such an objective, an experiment has been constructed to test a GA that uses it against one that does not. In order to judge the outcome of the experiment, the following research questions have been derived:

RQ1: Does a multi-objective solution using a priority objective and a quality objective give an improvement in quality?

RQ2: Does a multi-objective solution using a priority objective and a quality objective prioritize classes better than a solution that does not use the priority objective?

In order to address the research questions, the experiment will run a set of tasks to compare a default mono-objective set up to refactor a solution towards quality with a multi-objective approach that uses a quality objective and the newly proposed priority objective. The following hypotheses have been constructed to measure success in the experiment:

H1: The multi-objective solution gives an improvement in the quality objective value.

$H1_0$: The multi-objective solution gives no improvement in the quality objective value.

H2: The multi-objective solution gives significantly higher priority objective values than the corresponding mono-objective solution.

$H2_0$: The multi-objective solution does not give significantly higher priority objective values than the corresponding mono-objective solution.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the MultiRefactor tool used to conduct the experimentation. Section 4 explains the setup of the experiment used to test the priority objective, as well as the outcome of previous experimentation done to derive the quality objective and the GA parameters used. Section 5 discusses the results of the experiment by looking at the objective values and the times taken to run the tasks, and Section 6 concludes the paper.

2 RELATED WORK

Several recent studies in SBSM have explored the use of multi-objective techniques. Ouni, Kessentini, Sahraoui and Hamdi (Ouni et al. 2012) created an approach to measure semantics preservation in a software program when searching for refactoring options to improve the structure, by using the NSGA-II search. Ouni, Kessentini, Sahraoui and Boukadoum (Ouni et al. 2013) expanded upon the code smells correction approach of Kessentini, Kessentini and Erradi (Kessentini et al. 2011) by replacing the GA used with NSGA-II. Wang, Kessentini, Grosky and Meddeb (Wang et al. 2015) also expanded on the approach of Kessentini, Kessentini and Erradi by combining the detection and removal of software defects with an estimation of the number of future code smells generated in the software by the refactorings. Mkaouer et al. (Mkaouer et al. 2014; M. W. Mkaouer et al. 2015; W. Mkaouer et al.

2015) used NSGA-III to experiment with automated maintenance.

3 MultiRefactor

The MultiRefactor approach[1] uses the RECODER framework[2] to modify source code in Java programs. RECODER extracts a model of the code that can be used to analyze and modify the code before the changes are applied. MultiRefactor makes available various different approaches to automated software maintenance in Java programs. It takes Java source code as input and will output the modified source code to a specified folder. The input must be fully compilable and must be accompanied by any necessary library files as compressed jar files. The numerous searches available in the tool have various input configurations that can affect the execution of the search. The refactorings and metrics used can also be specified. As such, the tool can be configured in a number of different ways to specify the particular task that you want to run. If desired, multiple tasks can be set to run one after the other.

A previous study (Mohan et al. 2016) used the A-CMA (Koc et al. 2012) tool to experiment with different metric functions but that work was not extended to produce source code as an output (likewise, TrueRefactor (Griffith et al. 2011) only modifies UML and Ouni, Kessentini, Sahraoui and Boukadoum's (Ouni et al. 2013) approach only generates proposed lists of refactorings). MultiRefactor (Mohan and Greer 2017) was developed in order to be a fully-automated search-based refactoring tool that produces compilable, usable source code. As well as the Java code artifacts, the tool will produce an output file that gives information on the execution of the task including data about the parameters of the search executed, the metric values at the beginning and end of the search, and details about each refactoring applied. The metric configurations can be modified to include different weights and the direction of improvement of the metrics can be changed depending on the desired outcome.

MultiRefactor contains seven different search options for automated maintenance, with three distinct metaheuristic search techniques available. For each search type there is a selection of configurable properties to determine how the search will run. The refactorings used in the tool are mostly based on Fowler's list (Fowler 1999), consisting of 26 field-level, method-level and class-level refactorings, and are listed below.

Field Level Refactorings: Increase/Decrease Field Visibility, Make Field Final/Non Final, Make Field Static/Non Static, Move Field Down/Up, Remove Field.

Method Level Refactorings: Increase/Decrease Method Visibility, Make Method Final/Non Final, Make Method Static/Non Static, Remove Method.

Class Level Refactorings: Make Class Final/Non Final, Make Class Abstract/Concrete, Extract Subclass/Collapse Hierarchy, Remove Class/Interface.

The refactorings used will be checked for semantic coherence as part of the search, and will be applied automatically, ensuring the process is fully automated. A number of the metrics available in the tool are adapted from the list of metrics in the QMOOD (Bansiya and Davis 2002) and CK/MOOSE (Chidamber and Kemerer 1994) metrics suites. The 23 metrics currently available in the tool are listed below.

QMOOD Based: Class Design Size, Number Of Hierarchies, Average Number Of Ancestors, Data Access Metric, Direct Class Coupling, Cohesion Among Methods, Aggregation, Functional Abstraction, Number Of Polymorphic Methods, Class Interface Size, Number Of Methods.

CK Based: Weighted Methods Per Class, Number Of Children.

Others: Abstractness, Abstract Ratio, Static Ratio, Final Ratio, Constant Ratio, Inner Class Ratio, Referenced Methods Ratio, Visibility Ratio, Lines Of Code, Number Of Files.

In order to implement the priority objective, the important classes need to be specified in the refactoring tool (preferably by the developer(s) to express the classes they are most concerned about). With the list of priority classes and, optionally, non-priority classes and the list of affected classes in each refactoring solution, the priority objective score can be calculated for each solution. To calculate the score, the list of affected classes for each refactoring is inspected, and each time a priority class is affected, the score increases by one. This is done for every refactoring in the solution. Then, if a list of non-priority classes is also included, the affected classes are inspected again. This time, if a non-priority class is affected, the score decreases by 1. The higher the overall score for a solution, the more successful it is at refactoring priority classes and disfavoring non-priority classes. It is important to note that non-priority classes are not necessarily

[1] https://github.com/mmohan01/MultiRefactor

[2] http://sourceforge.net/projects/recoder

excluded completely but solutions that do not involve those classes will be given priority. In this way the refactoring solution is still given the ability to apply structural refactorings that have a larger effect on quality even if they are in undesirable classes, whereas the priority objective will favor the solutions that have applied refactorings to the more desirable classes.

4 EXPERIMENTAL DESIGN

In order to evaluate the effectiveness of the priority objective, a set of tasks were created that used the priority objective to be compared against a set of tasks that didn't. The control group is made up of a mono-objective approach that uses a function to represent quality in the software. The corresponding tasks use the multi-objective algorithm and have two objectives. The first objective is the same function for software quality as used for the mono-objective tasks. The second objective is the priority objective. The metrics used to construct the quality function and the configuration parameters used in the GAs are taken from previous experimentation on software quality. Each metric available in the tool was tested separately in a GA to deduce which were more successful, and the most successful were chosen for the quality function. The metrics used in the quality function are given in Table 1. No weighting is applied for any of the metrics. The configuration parameters used for the mono-objective and multi-objective tasks were derived through trial and error and are outlined in Table 2. The hardware used to run the experiment is outlined in Table 3.

For the tasks, six different open source programs are used as inputs to ensure a variety of different domains are tested. The programs range in size from relatively small to medium sized. These programs were chosen as they have all been used in previous SBSM studies and so comparison of results is possible. The source code and necessary libraries for all of the programs are available to download in the GitHub repository for the MultiRefactor tool. Each one is run five times for the mono-objective approach and five times for the multi-objective approach, resulting in 60 tasks overall. The inputs used in the experiment as well as the number of classes and lines of code they contain are given in Table 4.

For the multi-objective tasks used in the experiment, both priority classes and non-priority classes are specified for the relevant inputs. The number of classes in the input program is used to identify the number of priority and non-priority classes to specify, so that 5% of the overall number of classes in the input are specified as priority classes and 5% are specified as non-priority classes. In order to choose which classes to specify, the number of methods in each class of the input was found and ranked. The top 5% of classes that contain the most methods are the priority classes and the bottom 5% that contain the least methods are the non-priority classes for that input. Using the top and bottom 5% of classes means that the same proportion of classes will be used in the priority objective for each input program, minimizing the effect of the number of classes chosen in the experiment. In lieu of a way to determine the priority of the classes, their complexity as derived from the number of methods present, is taken to represent priority. Using this process, the configurations of the priority objective for each input were constructed and used in the experiment.

Table 1: Metrics used in the software quality objective, with corresponding directions of improvement for each.

Metrics	Direction
Data Access Metric	+
Direct Class Coupling	-
Cohesion Among Methods	+
Aggregation	+
Functional Abstraction	+
Number Of Polymorphic Methods	+
Class Interface Size	+
Number Of Methods	-
Weighted Methods Per Class	-
Abstractness	+
Abstract Ratio	+
Static Ratio	+
Final Ratio	+
Constant Ratio	+
Inner Class Ratio	+
Referenced Methods Ratio	+
Visibility Ratio	-
Lines Of Code	-

Table 2: GA configuration settings.

Configuration Parameter	Value
Crossover Probability	0.2
Mutation Probability	0.8
Generations	100
Refactoring Range	50
Population Size	50

The tool has been updated in order to use a heuristic to choose a suitable solution out of the final population with the multi-objective algorithm to inspect. The heuristic used is similar to the method used by Deb and Jain (Deb and Jain 2013) to construct a linear hyper-plane in the NSGA-III algo-

rithm. Firstly, the solutions in the population from the top rank are isolated and written to a separate sub folder. It is from this subset that the best solution will be chosen from when the task is finished. Among these solutions, the tool inspects the individual objective values, and for each, the best objective value across the solutions is stored. This set of objective values is the ideal point $\bar{z}=(z_1^{max}), (z_2^{max}), \ldots, (z_M^{max})$, where (z_i^{max}) represents the maximum value for an objective, and an objective $i = 1, 2, \ldots, M$. This is the best possible state that a solution in the top rank could have. After this is calculated, each objective score is compared with its corresponding ideal score. The distance of the objective score from its ideal value is found, i.e. $(z_i^{max})-f_i(x)$, where $f_i(x)$ represents the score for a single objective. For each solution, the largest objective distance (i.e. the distance for the objective that is furthest from its ideal point) is stored, i.e. $fmax(x)=max_{i=1}^{M}[(z_i^{max})-f_i(x)]$. At this point each solution in the top rank has a value, fmax(x), to represent the furthest distance among its objectives from the ideal point. The smallest among these values, $min_{j=0}^{N-1}$ fmax(x) (where N represents the number of solutions in the top rank), signifies the solution that is closest to that ideal point, taking all of the objectives into consideration. This solution is then considered to be the most suitable solution and is marked as such when the population is written to file. On top of this, the results file for the corresponding solution is also updated to mark it as the most suitable. This is how solutions are chosen among the final population for the multi-objective tasks to compare against the top mono-objective solution.

Table 3: Hardware details for the experimentation.

Operating System	Microsoft Windows 7 Enterprise Service Pack 1
System Type	64-bit
RAM	8.00GB
Processor	Intel Core i7-3770 CPU @ 3.40GHz

Table 4: Java programs used in the experimentation.

Name	**LOC**	**Classes**
Mango	3,470	78
Beaver 0.9.11	6,493	70
Apache XML-RPC 2.0	11,616	79
JHotDraw 5.3	27,824	241
GanttProject 1.11.1	39,527	437
XOM 1.2.1	45,136	224

For the quality function the metric changes are calculated using a normalization function. This function causes any greater influence of an individual metric in the objective to be minimized, as the impact of a change in the metric is influenced by how far it is from its initial value. The function finds the amount that a particular metric has changed in relation to its initial value at the beginning of the task. These values can then be accumulated depending on the direction of improvement of the metric (i.e. whether an increase or a decrease denotes an improvement in that metric) and the weights given to provide an overall value for the metric function or objective. A negative change in the metric will be reflected by a decrease in the overall function/objective value. In the case that an increase in the metric denotes a negative change, the overall value will still decrease, ensuring that a larger value represents a better metric value regardless of the direction of improvement. The directions of improvement used for the metrics in the experiment are given in Table 1. In the case that the initial value of a metric is 0, the initial value used is changed to 0.01 in order to avoid issues with dividing by 0. This way, the normalization function can still be used on the metric and its value still is low at the start. Equation (1) defines the normalization function, where $_m$ represents the selected metric, $_{C_m}$ is the current metric value and $_{I_m}$ is the initial metric value. $_{W_m}$ is the applied weighting for the metric (where 1 represents no weighting) and $_D$ is a binary constant (-1 or 1) that represents the direction of improvement of the metric. $_n$ represents the number of metrics used in the function. For the priority objective, this normalization function is not needed. The objective score depends on the number of priority and non-priority classes addressed in a refactoring solution and will reflect that.

$$\sum_{m=1}^{n} D.W_m \left(\frac{C_m}{I_m} - 1\right) \quad (1)$$

5 RESULTS

Figure 1 gives the average quality gain values for each input program used in the experiment with the mono-objective and multi-objective approaches. For most of the inputs, the mono-objective approach gives a better quality improvement than the multi-objective approach, although for Mango the multi-objective approach was better. For the multi-objective approach all the runs of each input were

able to give an improvement for the quality objective as well as look at the priority objective. For both approaches, the smallest improvement was given with GanttProject. The inputs with the largest improvements were different for each approach. For the mono-objective approach it was Beaver, whereas for the multi-objective approach, it was Apache XML-RPC.

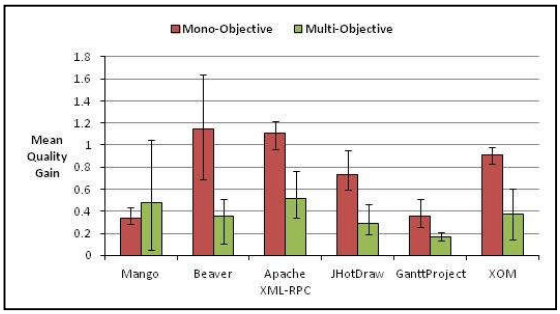

Figure 1: Mean quality gain values for each input.

Figure 2 shows the average priority scores for each input with the mono-objective and multi-objective approaches. For all of the inputs, the multi-objective approach was able to yield better scores coupled with the priority objective. The values were compared for significance using a one-tailed Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level (α = 5%). The priority scores for the multi-objective approach were found to be significantly higher than the mono-objective approach. For two of the inputs, Beaver and Apache XML-RPC, the mono-objective approach had priority scores that were less than zero. With the Beaver input, one of the runs gave a score of -6 and another gave a score of -10. Likewise, one run of the Apache XML-RPC input gave a priority score of -37. This implies that, without the priority objective to direct them, the mono-objective runs are less likely to focus on the more important classes (i.e. the classes with more methods), and may significantly alter the classes that should be disfavored (leading to the minus values for the three mono-objective runs across the two input programs).

Figure 3 gives the average execution times for each input with the mono-objective and multi-objective searches. For most of the input programs, the multi-objective approach took less time than the mono-objective but, for GanttProject, the multi-objective approach took longer. The Wilcoxon rank-sum test (two-tailed) was used again and the values were found to not be significantly different. The times for both approaches understandably increase as the input program sizes get bigger and the GanttProject input stands out as taking longer than the rest, although the largest input, XOM, is unexpectedly quicker. The execution times for the XOM input are smaller than both JHotDraw and GanttProject, despite it having more lines of code. However, both of these inputs do contain more classes. Considering the relevance of the list of classes in an input program to the calculation of the priority score, it makes sense that this would have an effect on the execution times. Indeed, GanttProject has by far the largest number of classes, at 437, which is almost double the amount that XOM contains. Likewise, the execution times for GanttProject are similarly around twice as large as those of XOM for the two approaches. The longest task to run was for the multi-objective run of the GanttProject input, at over an hour. The average time taken for those tasks was 53 minutes and 6 seconds.

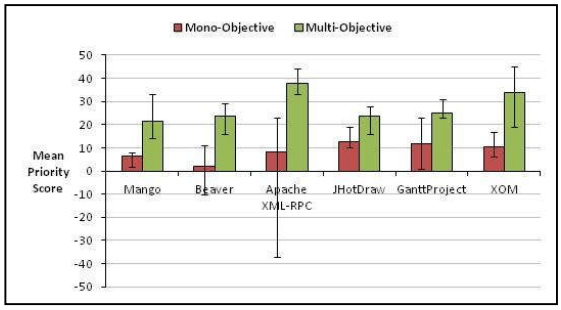

Figure 2: Mean priority scores for each input.

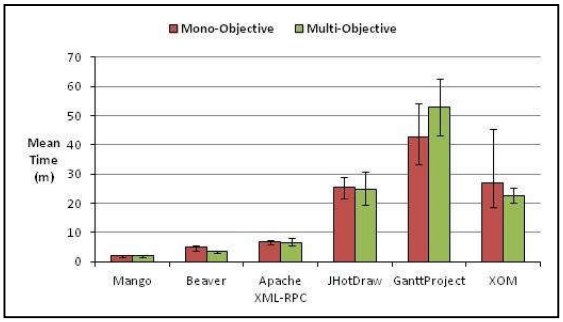

Figure 3: Mean times taken for each input.

6 CONCLUSIONS

In this paper an experiment was conducted to test a new fitness objective using the MultiRefactor tool. The priority objective measures the classes modified in a refactoring solution and gives an ordinal score that indicates the number of refactorings that relate to the important classes in the input program. These "priority classes" are specified as an extra input in

order for the program to calculate when the important classes are inspected. There is also an option to include a list of "non-priority classes" which, if refactored, will have a negative effect on the priority score. This objective helps the search to generate refactoring solutions that have focused on what a software developer envisions to be the more important areas of the software code, and away from other areas that should be avoided. The priority objective was tested in conjunction with a quality objective (derived from previous experimentation) in a multi-objective setup. To measure the effectiveness of the priority objective, the multi-objective approach is compared with a mono-objective approach using just the quality objective. The quality objective values are inspected to deduce whether improvements in quality can still be derived in this multi-objective approach and the priority scores are compared to measure whether the developed priority function can be successful in improving the focus of the refactoring approach.

The average quality improvement scores were compared across six different open source inputs and, for the most part, the mono-objective approach gave better improvements. The likely reason for the better quality score in the mono-objective approach is due to the opportunity for the mono-objective GA to focus on that single objective without having to balance the possibly contrasting aim of favoring priority classes and disfavoring non-priority classes. The multi-objective approach was able to yield improvements in quality across all the inputs. In one case, with the Beaver input, the multi-objective was able to not only yield an improvement in quality, but also generate a better improvement on average than the mono-objective approach. This may be due to the smaller size of the Beaver input, which could mean a restricted number of potential refactorings in the mono-objective approach. It could also be influenced by the larger range of results gained the multi-objective approach for that input. The average priority scores were compared across the six inputs and, for the mono-objective approach, were able to give some improvement. However, in some specific runs, the priority scores were negative. This would relate to there being more non-priority classes being refactored in a solution than priority classes, which, for the mono-objective approach, is unsurprising. The average priority scores for the multi-objective approach were better in each case. It is presumed that, as the mono-objective approach has no measures in place to improve the priority score of its refactorings, the solutions are more likely to contain non-priority classes and less likely to contain priority classes than the solutions generated with the multi-objective approach.

The average execution times for each input were inspected and compared for each approach. For most inputs, the multi-objective approach was slightly quicker than the mono-objective counterpart. The times for each input program increased depending on the size of the program and the number of classes available. The average times ranged from two minutes for the Mango program, to 53 minutes for GanttProject. While the increased times to complete the tasks for larger programs makes sense due to the larger amount of computation required to inspect them, XOM took less time than GanttProject and JHotDraw. Although XOM has more lines of code than these inputs, the reason more this is likely due to the number of classes available in each program, which is more reflective to the time taken to run the tasks for them. Therefore, it seems to be implied that the number of classes available in a project will have a more negative effect on the time taken to execute the refactoring tasks on that project than the amount of code. It was expected that, due to the higher complexity of the multi-objective GA in comparison to the basic GA, the execution times for the multi-objective tasks would be higher also. Although the times taken were similar for each approach, and were more affected by the project used, this wasn't the case for all of the inputs. This may have been due to the stochastic nature of the search. Depending on the iteration of the task run, there may be any number of refactorings applied in a solution. If one solution applied a large number of refactorings, this could likely have a noticeable effect on the time taken to run the task. The counterintuitive execution times between the mono-objective and multi-objective tasks may be a result of this property of the GA.

In order to test the aims of the experiment and derive conclusions from the results a set of research questions were constructed. Each research question and their corresponding set of hypotheses looked at one of two aspects of the experiment. **RQ1** was concerned with the effectiveness of the quality objective in the multi-objective setup. To address it, the quality improvement results were inspected to ensure that each run of the search yielded an improvement in quality. In all 30 of the different runs of the multi-objective approach, there was an improvement in the quality objective score, therefore rejecting the null hypothesis $\mathbf{H1_0}$ and supporting **H1**. **RQ2** looked at the effectiveness of the priority objective in comparison with a setup that did not use a function to measure priority. To address this, a non-

parametric statistical test was used to decide whether the mono-objective and multi-objective data sets were significantly different. The priority scores were compared for the multi-objective priority approach against the basic approach and the multi-objective priority scores were found to be significantly higher than the mono-objective scores, supporting the hypothesis **H2** and rejecting the null hypothesis $\mathbf{H2_0}$. Thus, the research questions addressed in this paper help to support the validity of the priority objective in helping to improve the focus of a refactoring solution in the MultiRefactor tool while in conjunction with another objective.

For future work, further experimentation could be conducted to test the effectiveness of the priority objective. The authors also plan to investigate other properties in order to create a better supported framework to allow developers to maintain software based on their preferences and their opinions of what factors are most important. It would also be useful to gauge the opinion of developers in industry and find out their opinion of the effectiveness of the MultiRefactor approach, and of the priority objective in an industrial setting.

ACKNOWLEDGEMENTS

The research for this paper contributes to a PhD project funded by the EPSRC grant EP/M506400/1.

REFERENCES

Bansiya, J. and Davis, C.G., 2002. A Hierarchical Model For Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering.*, 28(1), pp.4–17. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=979986.

Chidamber, S.R. and Kemerer, C.F., 1994. A Metrics Suite For Object Oriented Design. *IEEE Transactions on Software Engineering.*, 20(6), pp.476–493.

Deb, K. and Jain, H., 2013. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Non-Dominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation.*, 18(4), pp.1–23. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6600851%5Cnhttp://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6600851.

Fowler, M., 1999. *Refactoring: Improving The Design Of Existing Code*,

Griffith, I., Wahl, S. and Izurieta, C., 2011. TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility. In *24th International Conference on Computer Applications in Industry and Engineering, ISCA 2011.*

Kessentini, M., Kessentini, W. and Erradi, A., 2011. Example-Based Design Defects Detection And Correction. In *19th International Conference On Program Comprehension, ICPC 2011.* pp. 1–32.

Koc, E. et al., 2012. An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques. In E. Gelenbe, R. Lent, and G. Sakellari, eds. *Computer and Information Sciences II.* London: Springer London, pp. 59–66. Available at: http://link.springer.com/10.1007/978-1-4471-2155-8 [Accessed December 3, 2014].

Mkaouer, M.W. et al., 2015. On The Use Of Many Quality Attributes For Software Refactoring: A Many-Objective Search-Based Software Engineering Approach. *Empirical Software Engineering.*

Mkaouer, W. et al., 2014. High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives For Automating Software Refactoring Using NSGA-III. In *Genetic and Evolutionary Computation Conference, GECCO 2014.*

Mkaouer, W. et al., 2015. Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology.*, 24(3).

Mohan, M. and Greer, D., 2017. MultiRefactor: Automated Refactoring To Improve Software Quality. In *1st International Workshop on Managing Quality in Agile and Rapid Software Development Processes, QuASD 2017.* p. in press.

Mohan, M., Greer, D. and McMullan, P., 2016. Technical Debt Reduction Using Search Based Automated Refactoring. *Journal Of Systems And Software.*, 120, pp.183–194. Available at: http://dx.doi.org/10.1016/j.jss.2016.05.019.

Ouni, A. et al., 2013. Maintainability Defects Detection And Correction: A Multi-Objective Approach. *Automated Software Engineering.*, 20(1), pp.47–79.

Ouni, A. et al., 2012. Search-Based Refactoring: Towards Semantics Preservation. In *28th IEEE International Conference on Software Maintenance, ICSM 2012.* pp. 347–356.

Wang, H. et al., 2015. On The Use Of Time Series And Search Based Software Engineering For Refactoring Recommendation. In *7th International Conference on Management of computational and collective intElligence in Digital EcoSystems, MEDES 2015.* pp. 35–42.